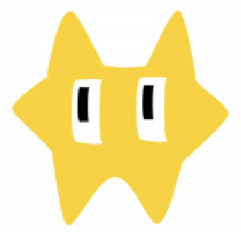
How to make a 2d Platform Game - part 1



Hello, and welcome back to my blog!

Its been a long time coming, but finally I've managed to get some time to work on the blog again...

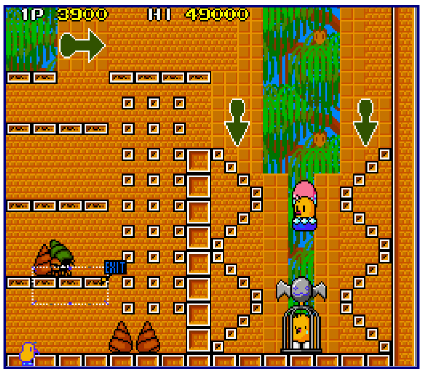
In this first instalment of a series blog articles, I'm going to be explaining step by step how you make this 2d platform game:

Click the game to give it focus... Apologies for the programmer art, and my level design (not my best qualities!)

The language is actionscript 3.0, but the techniques are applicable to all languages.

Inspiration

As a kid I always used to love playing The Newzealand Story and Rainbow Islands by Taito



The NewZealand Story

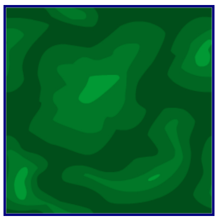
Rainbow Islands

And of late my love of platformers was rekindled by this awesome work in progress Generic:

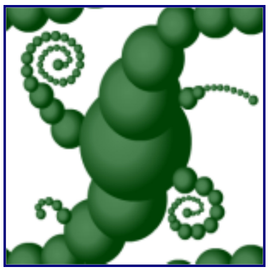
So I decided to write about the process of making one - you'll see influences from all three of these games in my bad graphics!

The background

Ok, one of the first things you will notice is that the game has a couple of layers of parallax in the background; this was relatively easy to achieve with the aid of a couple of tiles.



Background Tile



Mid-ground tile

Notice how the tiles are different sizes? This is important to stop the seams lining up too much as the camera moves about.

The most difficult thing was actually getting these designs to tile seamlessly when stacked next to each other, but that's because I'm a programmer not an artist! Anyway, I'll cover how I did that later when I talk about creating the tile set for the game.

Tiles follow camera

In order to get a seemingly infinite tiling background, the tiles are laid down by the renderer as they are needed, Wallace and Gromit style, as the camera moves around the level.

Gromit lays down the track just in time

In the picture above Gromit is laying down the track as the train moves him along; if you replace Gromit with the renderer and the train with the camera you should be able to see what I mean. The difference is that at the other end of the train, we would need another Gromit facing in the opposite direction picking up the track as the train passes over it, and then handing it back to the Gromit at the front!

I found that I needed a cache of N=Screen Width/Tile width + 2 in the X and N=Screen Height/Tile height + 2 in Y number of tiles in order to keep the screen full constantly.

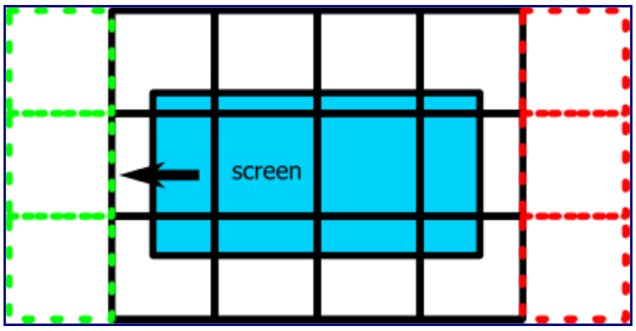


Figure 1

Figure 1 shows the screen in blue moving around a grid of tiles; green tiles are added at the front of the motion and red tiles are removed (or rather recycled) at the rear. Here is the code used:

```
package Code.Graphics { import flash.display.*; import Code.Geometry.*; import Code.Maths.*;
import Code.Constants; import Code.Platformer; public class TileRenderer { private var
m_tiles:Vector.<MovieClip>; private var m_numX:int; private var m_numY:int; private var
m_tileWidth:int; private var m_tileHeight:int; private var m_camera:Camera; private var
m_zDepth:Number; private var m_tempAabb:AABB; /// <summary> /// Constructor /// </summary>
public function TileRenderer( tileType:Class, width:int, height:int, camera:Camera,
stage:Platformer, zDepth:Number ) { m_tileWidth = width; m_tileHeight = height; m_camera =
camera; m_zDepth = zDepth; m_tempAabb = new AABB( ); m_numX =
Constants.kScreenDimensions.m_x/width+2; m_numY = Constants.kScreenDimensions.m_y/height+2;
m_tiles = new Vector.<MovieClip>( m_numX*m_numY ); // run though and create all the tiles we
need, this fuction takes // a closeure which actually does the work
PositionLogic( function( index:int, xCoord:int, yCoord:int ):void { m_tiles[index] = new
tileType( ); m_tiles[index].x = xCoord; m_tiles[index].y = yCoord;
m_tiles[index].cacheAsBitmap = true; // add the tile and send it to the back
stage.addChild( m_tiles[index] ); stage.setChildIndex( m_tiles[index], 0 ); }); } ///
<summary> /// This function runs through and computes the position of each tile - it takes a
closeure /// so you can insert your own inner logic to run at each location /// </summary>
private function PositionLogic( action:Function ):void
{ m_camera.GetWorldSpaceOnScreenAABB( m_tempAabb ); var screenTopLeft:Vector2 =
m_tempAabb.m_TopLeft; // stop the background from crawling around due to pixel trucation
screenTopLeft.RoundTo( ); // calculate the top left of the screen, scaled for z depth var
scaledTopLeft:Vector2 = screenTopLeft.MulScalar( 1/m_zDepth ); var tileX:int =
Math.floor(scaledTopLeft.m_x / m_tileWidth); var tileY:int = Math.floor(scaledTopLeft.m_y /
m_tileHeight); // this offset corrects for translation caused by the divide by z var
offset:Vector2 = scaledTopLeft.Sub( screenTopLeft ); // get the starting tile coords var
startX:int = tileX*m_tileWidth - offset.m_x; var startY:int = tileY*m_tileHeight -
offset.m_y; var xCoord:int = startX; var yCoord:int = startY; // run though and call the closure for each tile position for ( var j:int = 0; j<m_numY; j++ ) { xCoord = startX; for ( var i:int = 0; i<m_numX; i++ ) { var index:int = j*m_numX+i; action(index, xCoord, yCoord); xCoord += m_tileWidth; } yCoord += m_tileHeight; } } /// <summary> /// Update all the tiles to the new coordinates based on the camera's new position /// </summary> public function Update( ):void { PositionLogic( function( index:int, xCoord:int, yCoord:int ):void { m_tiles[index].x = xCoord; m_tiles[index].y = yCoord; }); } } }
```

A couple of important caveats with this technique:

In order to stop the tiles crawling around the screen very slightly due to coordinate trucation it was important to make sure the reference point for the top left of the screen was correctly rounded to an integer pixel boundary:

```
// stop the background from crawling around due to pixel trucation screenTopLeft.RoundTo( );
```

My function RoundTo() just rounds to the nearest integer.

Also, the parallax was achieved simply by dividing by the z value of the tile, and then correcting the position of the tiles so they started in the correct place on screen:

```
var scaledTopLeft:Vector2 = screenTopLeft.MulScalar( 1/m_zDepth ); var tileX:int = Math.floor(scaledTopLeft.m_x / m_tileWidth); var tileY:int = Math.floor(scaledTopLeft.m_y / m_tileHeight); var offset:Vector2 = scaledTopLeft.Sub( screenTopLeft );
```

So what's going on in the above code? What we're actually doing is taking the world-space position of the top left of the screen, dividing by the resolution of the tile to get the number of tiles required in each axis and then 'integerising' the result. Going back to the Gromit analogy, this stops Grommit from putting one piece of train track over the top of the last, or in front of it - he is forced to place them exactly one after the other, which means they tile without visual gaps. The last part is correcting for the offset which happens when we divide by the z depth of the tile layer - this forces the tiles to start at the top left of the screen.

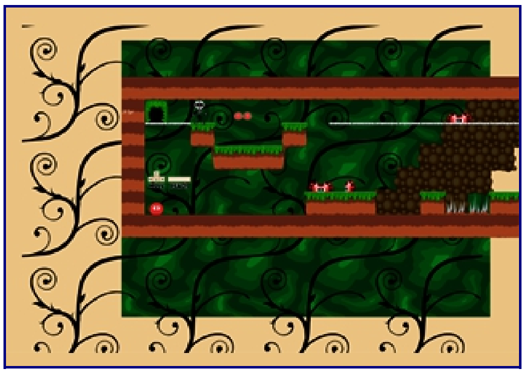


Figure 2

Figure 2 shows a grab of the game where you can see how the tiles are being placed outside the area normally visible to the camera.

The Tiles

Central to any old-school platform game are the tiles which make up the static part of the world. All the tiles in the game are a fixed size; 64x64 pixels, but you can obviously choose whatever tile size is most appropriate.

Tiles are multi-purpose in that they not only provide the building blocks from which every level is constructed, but that they also make up the collision geometry which the player interacts with as they play the level. They are also used to place down enemies and other special non-visible markers which affect certain AI behaviours.

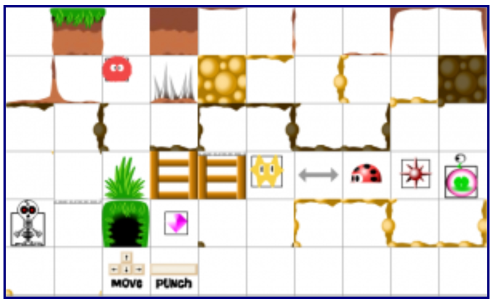


Figure 3

Figure 3 shows the complete tile-set from the game.

Each of the tiles above is assigned a unique integer which represents it, in this case starting from 0 and increasing left to right, top to bottom. The set of all of these integers for a particular level is called the Map. Maps must always be rectangular.

They are represented in code like this:

```
// map for the level private var m_map:Vector.<uint> = Vector.<uint>
( [ 04,04,04,04,04,04,04,04,04,04,04,04,04,04,04,04,04,04,
04,52,19,51,00,00,00,52,19,51,00,52,19,51,00,19,51,04,
04,17,15,18,00,00,00,17,15,18,35,17,15,49,51,15,18,04,
04,17,15,49,51,15,00,50,15,18,34,17,15,15,49,15,18,04,
04,00,48,15,49,15,50,15,47,29,34,17,15,48,15,15,18,04,
04,00,00,48,15,15,15,47,29,00,34,17,15,00,48,15,18,04,
04,00,00,00,16,16,16,29,00,00,00,00,16,00,31,16,29,04,
04,00,42,42,42,42,42,42,42,42,42,42,42,42,42,42,00,04,
04,00,13,00,00,00,00,00,00,00,00,00,00,00,00,00,00,04,
04,02,02,02,02,02,02,02,02,02,02,02,02,02,02,02,04 ] );
```

There is a big enum which allows me to identify which integer corresponds to which tile:

```
public class eTileTypes { // foreground static public const kEmpty:uint = 0; static public
const kEarthGrassLeft:uint = 1; static public const kEarthGrass:uint = 2; static public const
kEarthGrassRight:uint = 3; static public const kEarthMid:uint = 4; static public const
kEarthTop:uint = 5; static public const kEarthRight:uint = 6; static public const
kEarthLeft:uint = 7; static public const kEarthBottom:uint = 8; ... }
```

At level construction time, I run through the Map picking out each integer and constructing the relevant tile which represents it. The position of each tile in the world relates exactly to its position in the Map.

Tile coordinate system

The total extent of the world is defined as 64 x Nx, 64 x Ny, where Nx and Ny are the number of tiles in X and Y axis in the Map. I've also added an offset to locate 0,0 in pixel coordinates to the middle of the Map, in tile coordinates.

Tile coordinates are simply the tile indices into the map, and world coordinates are in pixels (in this case, since

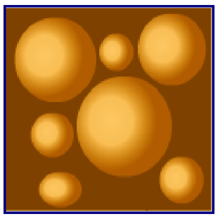
I'm using flash and that's most convenient). Its quite common to want to be able to convert from tile coordinates into world coordinates and vice versa, I do so like this:

```
/// <summary> /// calculate the position of a tile: 0,0 maps to
Constants.kWorldHalfExtents /// </summary> static public function
TileCoordsToWorldX( i:int ):Number { return i*Constants.kTileSize -
Constants.kWorldHalfExtents.m_x; } /// <summary> /// go from world coordinates to tile
coordinates /// </summary> static public function WorldCoordsToTileX( worldX:Number ):int
{ return ( worldX+Constants.kWorldHalfExtents.m_x )/Constants.kTileSize; }
```

There are corresponding versions for the other axis.

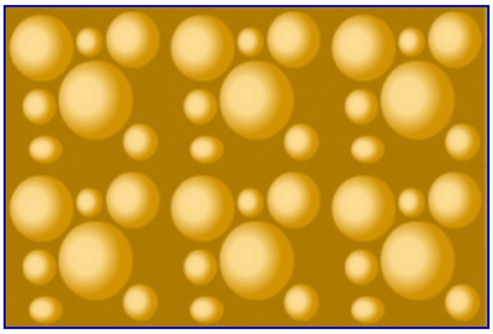
Tile psychology

How you actually design your tiles can have a massive effect on the number of them required to represent your game levels. Getting the tiles to seamlessly repeat is a quite irksome process if your not familiar with it.



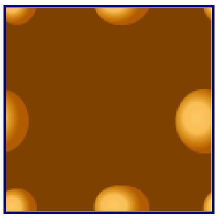
Cheese

Consider this cheese tile for example (which was actually designed to look like stone, but ended up like cheese due to my fantastic design skill). Looks nice enough by itself, but when you try tiling it a few times, obvious empty spaces emerge:



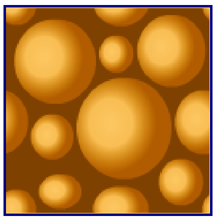
Tiling cheese

This is due to the spaces in the original design which were too small to fit a decent sized hole into. The solution is to design holes which are cut away on one side and continue on the other, like this:



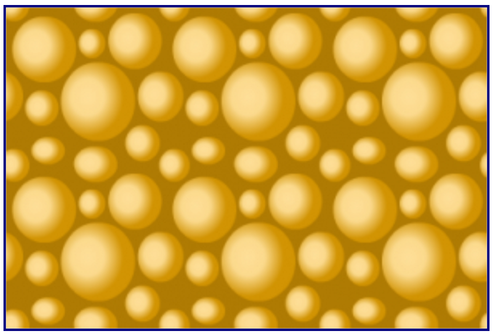
Side bits which tile

When combined with the original image we get this:



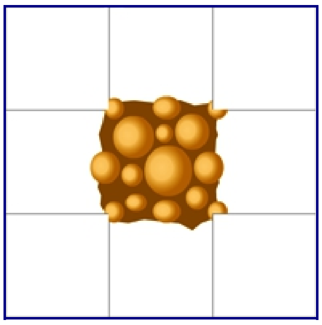
The final cheese

Which tiles much more pleasingly:



Final cheese tiling

But there is one important side effect which comes when designing the end pieces for each side of the tile - end pieces are tiles which represent the end of a piece of cheese, or earth and are there to make the design look less square and rigid.



Cheese plus 4 end pieces

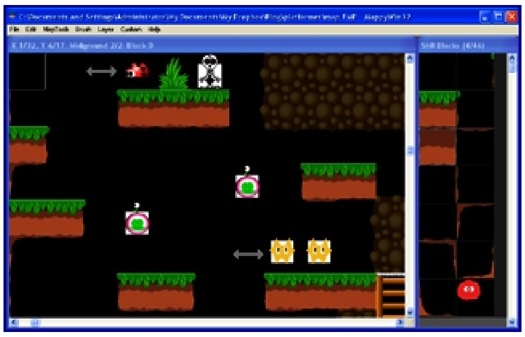
As you can see the end pieces help break up the squareness of the tile, but there is a problem. Due to the choice I made to tile the holes in the corners of the cheese, it means I not only need end pieces but also corner pieces to fill in the holes you can see above in each corner!

This is not only a lot more work when designing the tiles, its also more work when it comes to mapping them, because you need to map many more actual tiles to tidy up the design.

Mapping

In order to actually map down the tiles for each level you don't want to have to manually enter all the integers for each tile type, so I used Mappy instead, which is freely available and even has a script to output an AS3 array of data.

A level in Mappy looks like this:



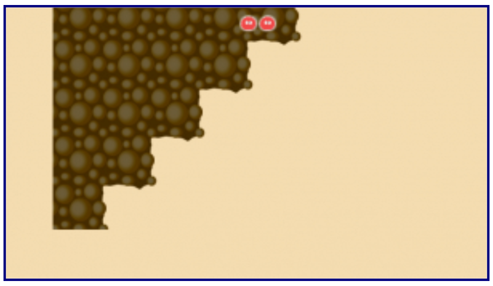
A level in mappy

You can see a selection of the tiles on the right and how they look once you've mapped them on the left.

Layers

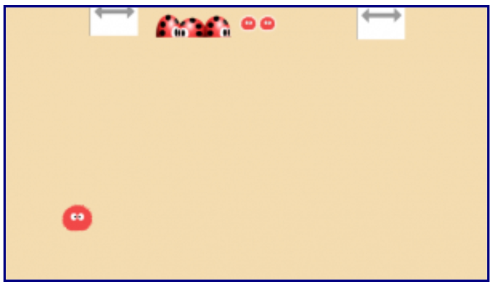
If you look carefully at the game, you'll notice that there isn't just one single layer of tiles for each level. There are actually three!

- The main foreground tiles, which consist of things the player collides with
- The mid-ground tiles which represent characters and special controls
- The background tiles, which are purely visual



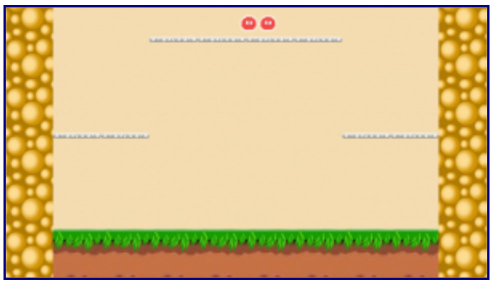
Background tiles only

I purposefully made all the background tiles much darker than the foreground so they don't look confusing to the player.



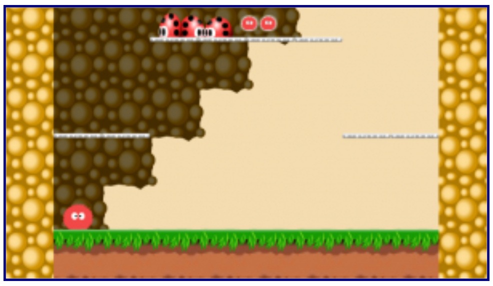
Mid-ground tiles only

The arrow markers you can see either side of the ladybirds actually control their motion - I'll talk more about this in a coming article.



Foreground tiles only

The foreground also acts as the collision layer - everything you see in this layer is collidable.



All layers together

Putting all there together, we get the above.

The reason to have three layers is that is gives you much greater flexibility when creating the levels; if you just used one, you wouldn't be able to map the player on a ladder, for example, because there would be only one tile slot in which to map both player and ladder.

It also adds a predefined depth sorting order - background tiles are always behind everything else, mid-ground tiles come after (although, most are not actually visible), and then the foreground tiles are top-most of all.

Having multiple layers doesn't add much complexity to the system and it gives a lot of flexibility.

Camera

The camera system I used here is much the same as the one I've described previously [in this article](), so I won't repeat myself.

I will just point out one improvement I made since last time: because we are using tiles, its of the utmost importance that there not be any visual cracking between the tiles as the player moves around the level. Such cracking would destroy the illusion of a continuous world. In order to prevent this, its very important that the camera only ever be positioned on whole pixel boundaries.

In order to achieve this I've added the following bit of code to the camera system:

```
// this is essential to stop cracks appearing between tiles as we scroll around - because
cacheToBitmap means // sprites can only be positioned on whole pixel boundaries, sub-pixel
camera movements cause gaps to appear. m_worldToScreen.tx =
Math.floor( m_worldToScreen.tx+0.5 ); m_worldToScreen.ty = Math.floor( m_worldToScreen.ty+0.5
);
```

m_worldToScreen is the matrix which ultimately gets attached to the main Sprite for the game, causing the world to be translated around as the player moves, giving the illusion that the player is moving around the world.

End of part 1

That's it for this instalment! Next time I'm going to starting talking about the collision detection of the player against the world, and also how the AI works.

As ever, if you want, you can buy the source-code for the entire game (or try a version **for free**), including all the assets and levels you see above. It will require Adobe Flash CS4+, the Adobe Flex Compiler 4.0+ and either Amethyst, or Flash Develop to get it to build. And you'll want Mappy or some alternative in order to create your own levels!

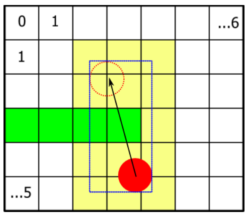
Following on from feedback from the Angry Birds article, I've included a Flash Develop project as well as an Amethyst project inside the .zip file, to help you get started more quickly, no matter which development environment you have.

You are free to use it for whatever purposes you see fit, even for commercial games or in multiple projects, the only thing I ask is that you don't spread/redistribute the source-code around. Please note that you will need some programming and design skills in order to make the best use of this!

Go to the source-code option page to choose the version you'd like - from **completely free** to the full version!

Subscribers can access the source here

How to make a 2d platform game - part 2 collision detection

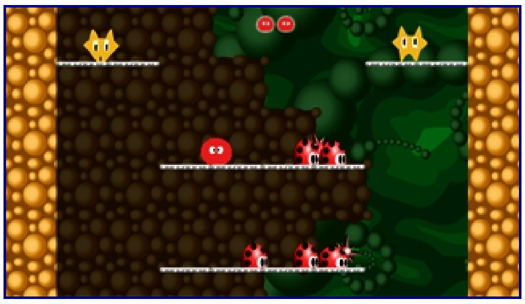


Hello and welcome back to my blog!

In this series of articles, I'm talking about the technology behind a platform game.

The language is actionscript 3.0, but the techniques are applicable to all languages.

In this particular article I'm going to talk about the physics, collision detection and AI aspects of the game.



The Game

There is a playable version at the bottom of the post, for those with Flash support in browser.

Class hierarchy

It makes sense at this point to talk about the class hierarchy I've used in the game, to represent everything which moves:

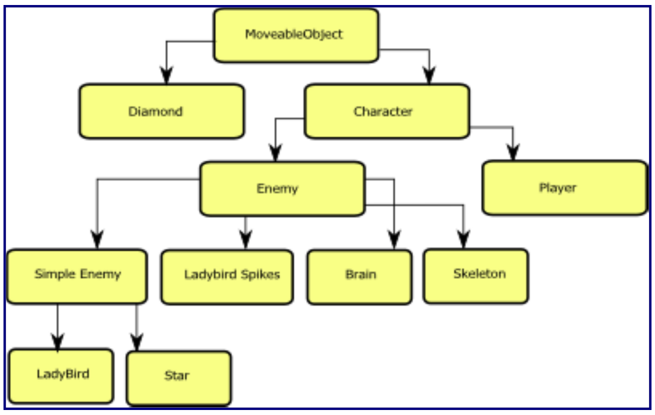


Figure 1

Figure 1 shows the class hierarchy - at the very top sits MoveableObject, which is where all the generic collision detection and response gets done; I say generic because the player does specialised work to handle things like ladders etc.

Each level in the hierarchy represents separate functionality; for example, Character contains the AnimationController which handles playing the various different animations for each character, SimpleEnemy represents a certain class of enemy character which does no collision detection with the world, and obeys special position markers when it encounters them. The Diamond pickup is a simple object which has no AI, and just collides with the world, so it inherits from the base class, since that's all the functionality it needs.

This may seem like a lot of extra complexity for such a simple game, but it really makes adding new enemy types very easy indeed, and simplifies the debugging process because there is so much code shared between each object.

If I had to give one piece of advice from 10 years of game development it would be this: avoid code duplication at all costs. It leads to slow development and bug city whereby you fix a bug in one location, and then forget to fix it in the other duplicated locations.

MoveableObject has certain properties which it requires be implemented by any class which inherits from it:

- m_HasWorldCollision - whether full collision detection should be performed
- m_ApplyGravity - whether gravity should be applied
- m_ApplyFriction - whether the world collision should apply friction

That way, and child class can chose what elements of collision detection it wants enabled.

Consider the following snippet from the Skeleton character:

```
package Code.Characters { import Code.Maths.Vector2; public class Skeleton extends Enemy
{ ... /// <summary> /// Apply collision detection only when not hurt /// </summary> public
override function get m_HasWorldCollision( ):Boolean { return !IsHurt(); } /// <summary> ///
Apply gravity only when not hurt /// </summary> protected override function get
m_ApplyGravity( ):Boolean { return !IsHurt(); } /// <summary> /// Apply friction only when
not hurt /// </summary> protected override function get m_ApplyFriction( ):Boolean { return !
IsHurt(); } ... } }
```

Its been set up to only do collision detection, apply gravity or friction when its not been 'hurt' (i.e punched by the player), this allows it to have the same behaviour as all other creatures when the player kills them.

Physics

Ok, so lets talk about the simple physics inside MoveableObject. Every MoveableObject has a position, a velocity and a radius.

This radius comes from the Flash IDE - an object instance called 'm_flaCollision' is checked for in the constructor of MoveableObject, and is a requirement, so the physics engine knows what the object's radius is. This radius is then turned into an AABB of size radius x radius, because the collision shapes are all AABBs.

Position is the position of the object in world space (i.e. pixels) and velocity is pixels/second. The update loop for MoveableObject looks like this:

```
package Code.Physics { import flash.display.*; import Code.Maths.Vector2; import Code.*;
import Code.System.*; import Code.Geometry.*; import Code.Graphics.*; import Code.Level.*;
public class MoveableObject extends MovieClip implements IAABB, ICircle { ... ///
<summary> /// Apply gravity, do collision and integrate position /// </summary> public
function Update( dt:Number ):void { if ( m_ApplyGravity )
{ m_vel.AddYTo( Constants.kGravity ); // clamp max speed m_vel.m_y = Math.min( m_vel.m_y,
Constants.kMaxSpeed*2 ); } if ( m_HasWorldCollision ) { // do complex world collision
Collision( dt ); } // integrate position m_pos.MulAddScalarTo( m_vel.Add(m_posCorrect), dt );
// force the setter to act m_Pos = m_pos; m_posCorrect.Clear( ); } ... } }
```

This is the part which actually calls out to the various features of MoveableObject and depends on the child class's implementation of those properties I discussed earlier. Child classes call out to this function from their own Update() which contains their specific logic.

This is just your basic physics set-up:

- Add gravity
- Do collision detection
- Integrate position

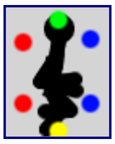
This is basically all we need for the game as it stands.

Collision detection

When writing the demo above, I did a lot of research on collision detection techniques for platform games because I knew that back in the 1980s when these games first came out, there were no floating point units and collision detection research was in its infancy, so there must be some really easy tricks you can do to get a great result nice and simply.

Update: I wish I'd found this article before I started writing this, its about how the original developers of M.C.Kids on the SNES handled tile based collision detection. In summary its kind of similar to the article I reference below, in that it used collision points, only it goes into more detail about all the ins and outs of the technique.

This article contains the most detailed explanation that I could find, but on implementing it I found there were a few things I didn't like; the author advocates using a number of predefined points around the player to help detect collisions and to judge what to do next, like this:



Points around player

The author writes:

If the point we check on top goes inside a solid block, we move the player downward so that the top point is just below the block it bumped into. If either right point goes inside a solid block, we move the player left until the offending point is just left of the block it bumped into, and so on...

Another benefit of detecting player collision using six points in that hexagon configuration is that if the player is jumping horizontally and the feet hit a corner, the player is automatically bumped up onto the surface; if the player falling vertically hits a corner off-center, or steps off a ledge, the player slides off away from the wall. (Try this! It feels much better than it would if the player behaved as a boxy rectangle.)

However, I found that the last part he mentions about the player being automatically bumped onto the surface was very jarring and often left me wondering what had happened while playing the game. I also found that it didn't help me when the player was moving quickly and became embedded inside a block, because with all points inside the block, there was no clearly correct way to resolve. Also, the amount of code I found I needed was getting excessive so I decided to discard this method - maybe it would have been better if my player character was taller like in his example.

A new way and tiles as a broad-phase

Obviously I needed a new way, but first lets talk about the tile coordinate system and how it makes collision detection nice and easy.

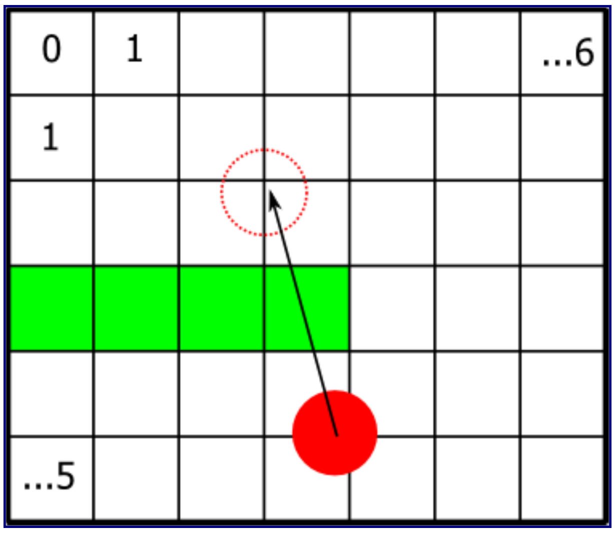


Figure 2

Consider Figure 2 in which the tile coordinates have been numbered 0-6 in the X axis, and 0-5 in the Y. This shows a typical scenario where the player character (shown in red) is jumping up and will hit the green platform at some point between the current frame and the next frame (the player's velocity is shown as the arrow). In order for the collision system to know which tiles need checking against the player, its a simple matter to enclose the player's range of motion within an axis aligned bounding box, or AABB.

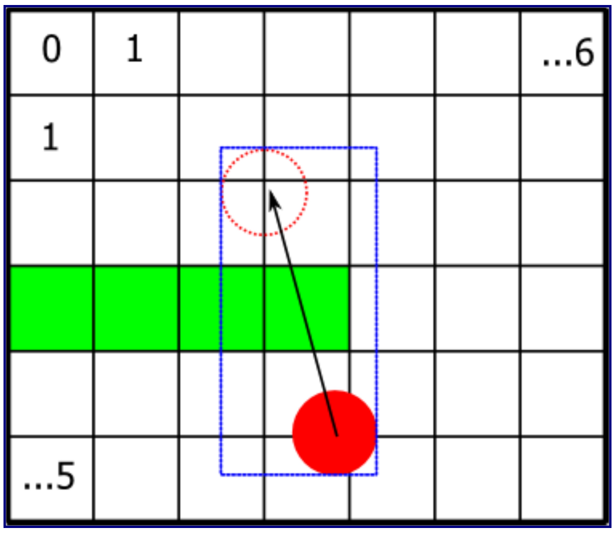


Figure 3

Figure 3 shows this bounding box overlaid on the scene in blue. If we take that bounding box and highlight every tile which intersects with it, we now know which tiles we need to consider for collision detection against the player.

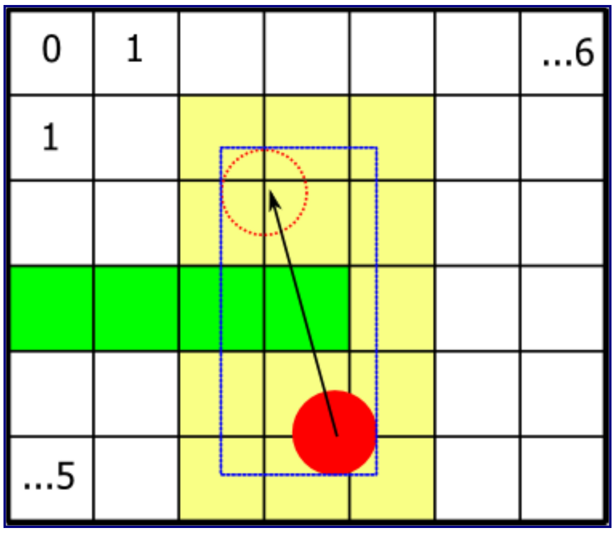


Figure 4

Figure 4 shows the relevant tiles highlighted in yellow.

Because there is a direct 1->1 mapping between world coordinates and tile coordinates, it becomes incredibly easy to get access to the tiles in order to perform fine grained collision detection against. This is in essence what a broad-phase collision detection system does, and its refreshing to see it arise naturally as a direct consequence of using a tile engine in the first place.

The new way

Because we don't want the player (or any other fast moving object) passing though platforms if they move too quickly, the fine grained collision detection system, or narrow phase, must be good enough to prevent this from happening.

I'm going to use a technique I first talked about a while back, called Speculative Contacts. Don't worry if it sounds horribily complex, its actually rather simple.

All it requires to work is a function which can return the distance between any two objects.

The collision shape

Before I go into the details, its important that I talk about how the choice of collision shape will affect the feel of the game. I started out with a circle to represent moving objects, but I soon realised this wasn't going to work.

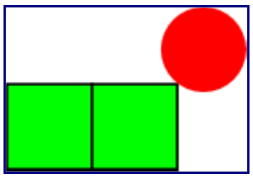


Figure 5

The reason is obvious looking at Figure 5; circles tend to roll smoothly off the edges of objects when placed right on them, which is exactly what you don't want happening when you're lining yourself up for a jump.

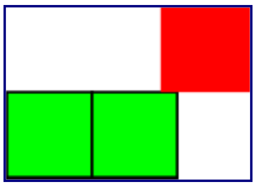


Figure 6

A better choice is the AABB, which naturally cannot rotate and therefore will allow objects to perch right on the edge of platforms without falling off them, as shown in Figure 6.

Distance function

In order to implement Speculative Contacts in this case, we need a distance function which will give us distance between two AABBs.

In order to achieve this we turn to the a technique from the Minkowski Difference. If we shrink one AABB down to a point, and grow the other one by the extents (width and height) of the first, the problem becomes one of finding the distance between the point and the new AABB.

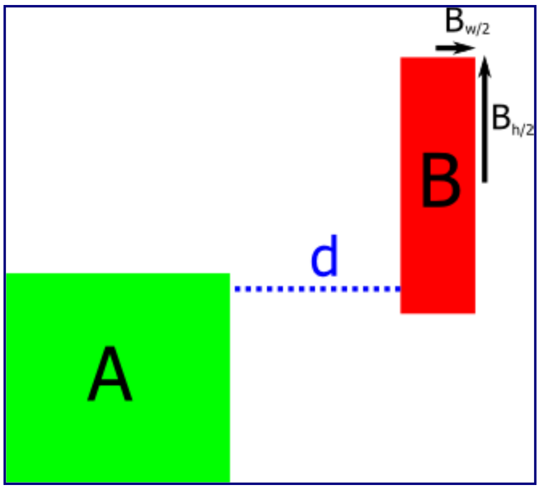


Figure 7

Figure 7: In order to find the distance d, between AABBs A and B...

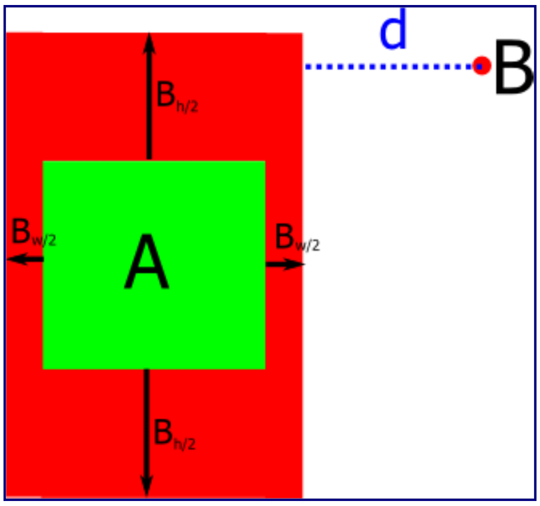


Figure 8

Figure 8: We shrink B down to a point and expand A by the extents of B, then we can use a simple distance from AABB to point function.

Distance from AABB to point

We actually only need the closest distance in each axis for our purposes, so we're ignoring the case where the corner of the AABB is the closest to the point.

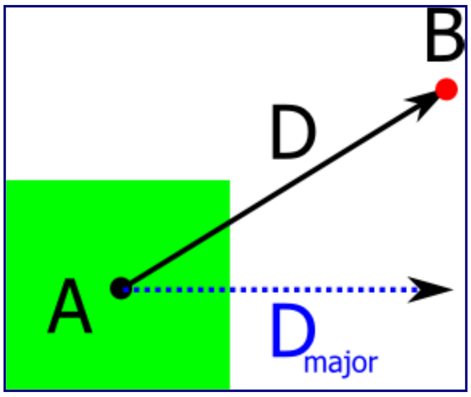


Figure 9

Figure 9: To find the distance between AABB A and point B we calculate the vector from A->B, D and then take the Major Axis of this vector. That is, the signed, unit length vector in which the only coordinate filled in represents the largest coordinate of the original vector.

```
/// <summary> /// Get the largest coordinate and return a signed, unit vector containing only
that coordinate /// </summary> public function get m_MajorAxis( ):Vector2 { if
( Math.abs( m_x )>Math.abs( m_y ) ) { return new Vector2( Scalar.Sign(m_x), 0 ); } else
{ return new Vector2( 0, Scalar.Sign(m_y) ); } }
```

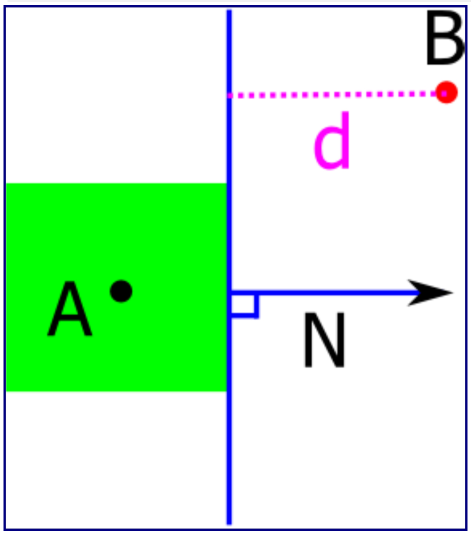


Figure 10

Figure 10: The Major Axis has now become the plane normal for our collision. We can calculate the position of the plane by scaling this normal by the half extents of A and adding on the position of A in world space). The distance d, from point B to this new plane is the final distance between point and AABB, and thus the distance between two AABBs.

Speculative Contacts

Now we have all the tools we need to get this technique working!

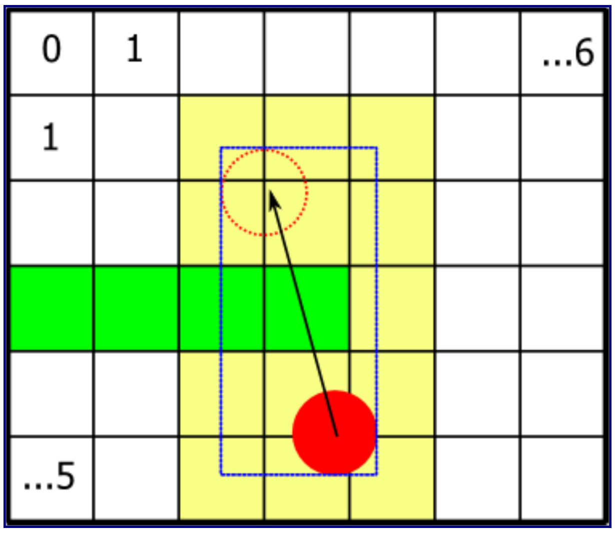


Figure 4

Looking back at Figure 4 again, all we have to do now is to query the map to see if any of those tiles highlighted in yellow are collidable and if so, we must calculate the distance to each one and the normal as we did above.

```
package Code.Physics { import flash.display.*; import Code.Maths.Vector2; import Code.*;
import Code.System.*; import Code.Geometry.*; import Code.Graphics.*; import Code.Level.*;
public class MoveableObject extends MovieClip implements IAABB, ICircle { ... ///
<summary> /// Do collision detection and response for this object /// </summary> protected
function Collision( dt:Number ):void { // where are we predicted to be next frame? var
predictedPos:Vector2 =
Platformer.m_gTempVectorPool.AllocateClone( m_pos ).MulAddScalarTo( m_vel, dt ); // find
min/max var min:Vector2 = m_pos.Min( predictedPos ); var max:Vector2 =
m_pos.Max( predictedPos ); // extend by radius min.SubFrom( m_halfExtents );
max.AddTo( m_halfExtents ); // extend a bit more - this helps when player is very close to
boundary of one map cell // but not intersecting the next one and is up a ladder min.SubFrom(
Constants.kExpand ); max.AddTo( Constants.kExpand ); PreCollisionCode( );
m_map.DoActionToTilesWithinAabb( min, max, InnerCollide, dt ); PostCollisionCode( ); } ///
<summary> /// Inner collision response code /// </summary> protected function
InnerCollide(tileAabb:AABB, tileType:int, dt:Number, i:int, j:int ):void { // is it
collidable? if ( Map.IsTileObstacle( tileType ) ) { // standard collision responce var
collided:Boolean = Collide.AabbVsAabb( this, tileAabb, m_contact, i, j, m_map ); if
( collided ) { CollisionResponse( m_contact.m_normal, m_contact.m_dist, dt ); } } else if
( Map.IsJumpThroughPlatform( tileType ) || Map.IsTileLadderTop(tileType) ) { // these type of
platforms are handled separately since you can jump through them collided =
Collide.AabbVsAabbTopPlane( this, tileAabb, m_contact ); if ( collided ) { CollisionResponse(
m_contact.m_normal, m_contact.m_dist, dt ); } } } ... } }
```

The above snippet shows the relevant code in the MoveabeObject class. The function Collision is the entry point.

```
package Code.Level { import Code.Geometry.AABB; import Code.System.*; import
Code.Maths.Vector2; import Code.*; public class Map { ... /// <summary> /// Call out to the
action for each tile within the given world space bounds /// </summary> public function
DoActionToTilesWithinAabb( min:Vector2, max:Vector2, action:Function, dt:Number ):void { //
round down var minI:int = WorldCoordsToTileX(min.m_x); var minJ:int =
WorldCoordsToTileY(min.m_y); // round up var maxI:int = WorldCoordsToTileX(max.m_x+0.5); var
maxJ:int = WorldCoordsToTileY(max.m_y+0.5); for ( var i:int = minI; i<=maxI; i++ ) { for
( var j:int = minJ; j<=maxJ; j++ ) { // generate aabb for this tile FillInTileAabb( i, j,
m_aabbTemp ); // call on the mid-ground map (ladders and special objects) action( m_aabbTemp,
GetMidgroundTile( i, j ), dt, i, j ); } } for ( i = minI; i<=maxI; i++ ) { for ( j = minJ;
j<=maxJ; j++ ) { // generate aabb for this tile FillInTileAabb( i, j, m_aabbTemp ); // call
the delegate on the main collision map action( m_aabbTemp, GetTile( i, j ), dt, i, j ); } } }
... } }
```

The above snippet shows the code which actually does the looping over the tiles in the map shown in Figure 4. You can see that I do two loops, one for the mid-ground tiles and one for the foreground. This is primarily because of ladders which are currently mapped in the mid-ground layer. I would like to explore mapping them solely in foreground in a future version, though as it would make the code smaller.

The interesting thing about Speculative Contacts is that they do no work if they're not required to - so, for all the yellow tiles in Figure 4 we calculate the distance to each one and the normal at that point, but the function CollisionResponse doesn't do anything unless the following condition is true:

```
var nv:Number = m_vel.Dot( normal ) + separation/dt; if (nv < 0) { // do something }
```

What this is saying is: if the projection of the velocity of the object onto the contact normal (i.e. the velocity in the normal direction) is less than the closest distance between the objects (divided by the timestep, to convert to the same units), then do nothing. I.e. if the objects can not touch between this frame and next, do nothing.

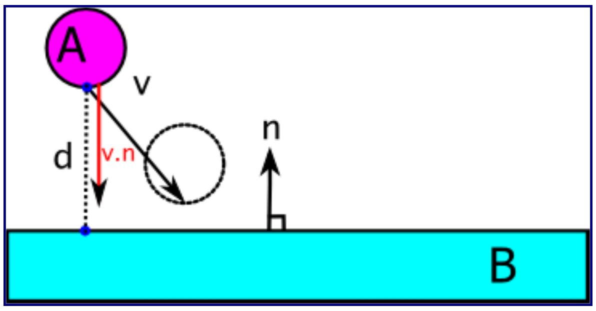


Figure 11

Figure 11 show this in diagram form, A is heading towards B and the projected velocity is shown (v.n) but is shorter than the distance between objects d and so no collision is possible.

There is one important caveat to mention, in that if the objects are moving fast enough, ghost collisions are possible - whereby one object will appear to hit an object that it shouldn't have. This occurs because the CollisionResponse code only knows about the infinite plane of the collision point and not the actual geometry (for performance reasons), so a fast moving object will sometimes hit the empty space next to the surface of the object rather than passing through. However, this is not a noticeable artefact in the game shown on this page, particularly because the maximum speed of all objects is clamped.

Internal edges

One of the problems that a lot of collision detection systems face is that of internal edges and unfortunately, this game is no different.

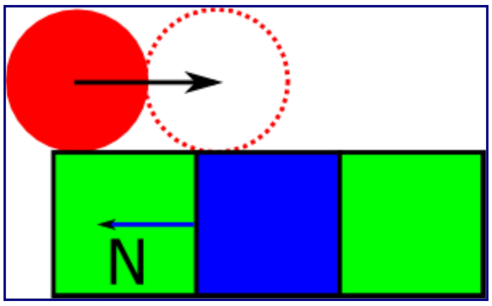
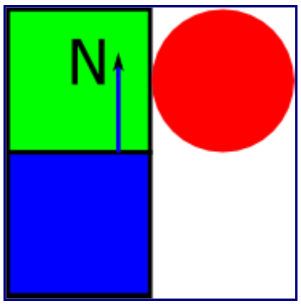


Figure 12

In Figure 12, the red ball will be judged for collision against the blue block, and the distance function will return the unit vector N as the closest direction, this is correct in isolation, but considering there are other blocks on either side and this is in fact one continuous surface, problems arise. If nothing is done to prevent this, the player will get stuck on the gap between blocks and will even be able to stand the gaps between vertical blocks.



Player stands on the gap between blocks

Solution

To prevent this from happening, I have some special code in the collision detection system which checks for internal edges.

```
package Code.Geometry { import Code.System.*; import Code.Maths.*; import Code.Geometry.*; import Code.Characters.Character; import Code.Level.*; import Code.eTileTypes; import Code.Platformer; import Code.Constants; public class Collide { /// <summary> /// Helper function which checks for internal edges /// </summary> static private function IsInternalCollision( tileI:int, tileJ:int, normal:Vector2, map:Map ):Boolean { var nextTileI:int = tileI+normal.m_x; var nextTileJ:int = tileJ+normal.m_y; var currentTile:uint = map.GetTile( tileI, tileJ ); var nextTile:uint = map.GetTile( nextTileI, nextTileJ ); var internalEdge:Boolean = Map.IsTileObstacle(nextTile); return internalEdge; } ... } }
```

What this does is to simply check the map to see if there is another collide-able tile 'next to' the current one. 'Next to' is defined as one tile along from the current one in the direction of the collision normal. If there is a tile there which is an obstacle, this collision is marked as an internal edge and the system discards it.

Going back to Figure 12, the block which is checked is the one pointed to by the normal, this is a collide-able block and so the collision is discarded correctly.

Jump-through platforms

Some types of platform game are built of completely solid platforms which you cannot jump through, but others like The Newzealand Story and Rainbow Islands are built entirely of platforms that the player can jump through.

To make this game engine as flexible as possible I wanted to support both.

In order to achieve this I needed a distance function which would give me the distance between an point and just the top plane of another (remember we shrunk down one AABB to a point and grew the other). Even still, I only want to consider colliding with this if the moving object was sufficiently close to the top plane of the AABB - if not, the collision is discarded.

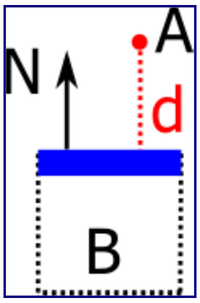


Figure 14

Figure 14 shows jump-through platform B and movable object A, which is d distance to the top plane of B and N is the major axis.

There are two collision acceptance conditions on top of the regular AABB vs AABB code.

- If the major axis is pointing up, i.e. this is a ground platform
- If the distance of the moving object to the top of the platform is greater than some negative tolerance

If these conditions are satisfied then the collision is accepted. If not, its rejected. The negative tolerance is to handle the transition between being just under the top of the platform and being on it. Its negative because when the moving object is below the platform, the distance is always negative; it grows less and less negative as the object moves up until its 0 right on the platform, then it becomes positive again. This tolerance is shown as the blue bar in Figure 14.

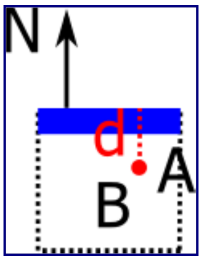


Figure 15

Figure 15 shows one discarded case, where the moving object is too far below the blue bar of tolerance.

Collision Response

The collision response at work here is just to remove the normal velocity of the moving object completely upon collision (and also correct for any penetration which occurs). This means objects do not bounce, but for this game, that is an acceptable compromise.

```
package Code.Physics { import flash.display.*; import Code.Maths.Vector2; import Code.*;
import Code.System.*; import Code.Geometry.*; import Code.Graphics.*; import Code.Level.*;
public class MoveableObject extends MovieClip implements IAABB, ICircle { ... ///
<summary> /// Collision Reponse - remove normal velocity /// </summary> protected function
CollisionResponse( normal:Vector2, dist:Number, dt:Number ):void { // get the separation and
penetration separately, this is to stop pentration // from causing the objects to ping apart
var separation:Number = Math.max( dist, 0 ); var penetration:Number = Math.min( dist, 0 ); //
compute relative normal velocity require to be object to an exact stop at the surface var
nv:Number = m_vel.Dot( normal ) + separation/dt; // accumulate the penetration correction, this is applied in Update() and ensures // we don't add any energy to the system m_posCorrect.SubFrom( normal.MulScalar( penetration/dt ) ); if ( nv<0 ) { // remove normal velocity m_vel.SubFrom( normal.MulScalar( nv ) ); // is this some ground? if ( normal.m_y<0 ) { m_onGround = true; // friction if ( m_ApplyFriction ) { // get the tanget from the normal (perp vector) var tangent:Vector2 = normal.m_Perp; // compute the tangential velocity, scale by friction var tv:Number = m_vel.Dot( tangent )*kGroundFriction; // subtract that from the main velocity m_vel.SubFrom( tangent.MulScalar( tv ) ); } if (!m_onGroundLast) { // this transition occurs when this object 'lands' on the ground LandingTransition( ); } } } } ... } }
```

The first few lines compute the separation and penetration (positive or negative components of the distance between objects)

```
// get the separation and penetration separately, this is to stop pentration // from causing the objects to ping apart var separation:Number = Math.max( dist, 0 ); var penetration:Number = Math.min( dist, 0 );
```

Then, I compute the relative normal velocity and accumulate the position correction vector which handles penetration resolution:

```
// compute relative normal velocity require to be object to an exact stop at the surface var nv:Number = m_vel.Dot( normal ) + separation/dt; // accumulate the penetration correction, this is applied in Update() and ensures // we don't add any energy to the system m_posCorrect.SubFrom( normal.MulScalar( penetration/dt ) );
```

If the normal velocity is less than 0, i.e the objects will touch between this frame and next, remove the normal velocity from the moving object:

```
if ( nv<0 ) { // remove normal velocity m_vel.SubFrom( normal.MulScalar( nv ) );
```

If this is a piece of ground we're going to collide with, record that fact, and then if we need to apply friction, do so:

```
m_onGround = true; // friction if ( m_ApplyFriction ) { // get the tanget from the normal (perp vector) var tangent:Vector2 = normal.m_Perp; // compute the tangential velocity, scale by friction var tv:Number = m_vel.Dot( tangent )*kGroundFriction; // subtract that from the main velocity m_vel.SubFrom( tangent.MulScalar( tv ) ); }
```

If we weren't recorded as being on the ground in the last collision, then call out to some code which handles landing on the ground (this is user definable code):

```
if (!m_onGroundLast) { // this transition occurs when this object 'lands' on the ground LandingTransition( ); }
```

And that's it! The only slightly hairy part is the friction calculation, but even that is relatively simple - it just gets the unit length vector perpendicular to the normal (i.e. the sliding vector), calculates the object's velocity in the sliding direction, scales that down by some friction coefficient and then subtracts that from the total velocity.

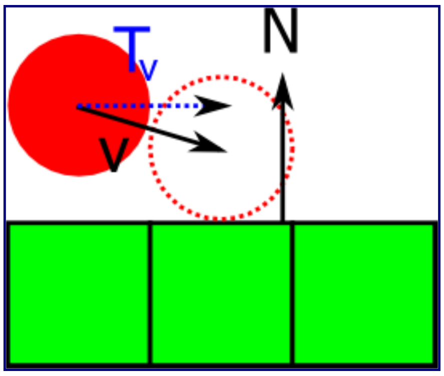


Figure 13

Figure 13 shows the computation of the tangential velocity; if you were to remove this tangential velocity from the full velocity you would have infinite friction.

If you would like more in-depth details of the collision response code, I suggest you read Physics Engines For Dummies, which covers this in great detail.

End of part 2

Wow, I thought I would have space to talk about the AI in this article, but obviously not! There was an unexpectedly large amount of collision detection to discuss. Ok, so next time I will definitely talk more about the AI and the software engineering behind it all.

As ever, if you want, you can buy the source-code for the entire game (or even try a version **for free**), including all the assets and levels you see above. It will require Adobe Flash CS4+, the Adobe Flex Compiler 4.0+ and either Amethyst, or Flash Develop to get it to build. And you'll want Mappy or some alternative in order to create your own levels!

Following on from feedback from the Angry Birds article, I've included a Flash Develop project as well as an Amethyst project inside the .zip file, to help you get started more quickly, no matter which development environment you have.

You are free to use it for whatever purposes you see fit, even for commercial games or in multiple projects, the only thing I ask is that you don't spread/redistribute the source-code around. Please note that you will need some programming and design skills in order to make the best use of this!

Click the game to give it focus... Apologies for the programmer art, and my level design (not my best qualities!)

Go to the source-code option page to choose the version you'd like - from **completely free** to the full version!

Subscribers can access the source here

How to make a 2d Platform Game - part 3 ladders and AI

Hello and welcome back to my blog!

This is part 3 in a series of articles where I talk about how to go about making a 2d platform game like this one:

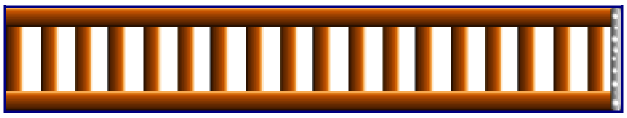
In this particular episode I'm going to talk about the horror of Ladders and the joy of AI.

Ladders

Ok, so ladders are a complete pain in the ass on all fronts! The reason being is that they represent a discontinuity in control mechanism and physics response.

The player must transition from walking along in an environment where they experience gravity and can jump to one where they do not and cannot jump. On top of that, you have several different methods of starting the climb:

- Walking along the floor and joining the bottom of a ladder
- Walking along the top of a platform and joining the top of a ladder
- Landing on the middle of the ladder after falling or doing a jump



The Horror

So why have these distinctions?

Walking along the floor and joining the bottom of a ladder

You don't want to walk past the bottom of a ladder and automatically start climbing it, so that means you must have a special case for being at the bottom of the ladder and pressing up (which would normally have jumped you).

Walking along the top of a platform and joining the top of a ladder

Again, you don't want to start climbing down a ladder just by walking over the very top of one, or indeed by landing on the top, so you need a special case for pressing down while being on the top of the ladder. Additionally, this is made even more of a pain because this is the only instance in the game where the player can pass down through a platform (in order to join the ladder).

Landing on the middle of the ladder after falling or doing a jump

I did wrestle with this idea and for a while I had it set so you couldn't land on the middle of a ladder but after a while of playing I realised this was going to be too fiddly for the player and might lead to a bad play experience. Landing on the middle of the ladder is tricky because it has to act exactly like a platform, but a platform which exists at whatever pixel you land. Also, it represents yet another transition from normal falling to ladder climbing.

Ladder tile types

I have two different tiles which represent a ladder:

The middle

The middle section and:

The top

The top part. The top part is special since it acts like a jump through platform and a ladder, depending on where the player is in relation to it.

Solving the problem

In order to address each of these issues I found I needed to track a fair amount of state and transitions between state.

```
package Code.Characters { ... public class Player extends Character { ... private var m_collideLadder:uint; private var m_oldCollideLadder:uint; private var m_climbLadder:Boolean; private var m_onLadderTop:Boolean; private var m_disableLadderTopCollision:Boolean; ... } }
private var m_collideLadder:uint;
```

The variable m_collideLadder is updated every frame and represents the ladder tile type that the player is currently colliding with (i.e. player's centre point is within the AABB of the tile). It can either be eTileTypes.kLadderMid, eTileTypes.kLadderTop, or eTileTypes.kInvalid when not colliding with a ladder tile.

In addition I have:

```
private var m_oldCollideLadder:uint;
```

This tracks the last state of m_collideLadder, so whenvever m_collideLadder changes, the previous state is stored in m_oldCollideLadder. This allows me to detect transitions between colliding with the ladder and not.

In particular the case where the player lands on the middle of a ladder by falling or jumping looks like this:

```
package Code.Characters { ... public class Player extends Character { ... /// <summary> ///
Special implementation for the player /// </summary> protected override function
PostCollisionCode( ):void { super.PostCollisionCode( ); // if we're not colliding with the
ladder or we're on the ground, we're not climbing the ladder if ( !m_collideLadder ||
m_OnGround ) { m_climbLadder = false; } // if we fall onto the top of a ladder, we start
climbing it if ( m_oldCollideLadder==eTileTypes.kEmpty && Map.IsTileLadder(m_collideLadder)
&& !m_OnGround ) { m_climbLadder = true; m_animationController.PlayOnce( kLandAnim ); } if
( !m_OnGround&&m_OnGroundLast && m_vel.m_y > 0 )
{ m_animationController.PlayOnce( kFallAnim ); } } ... } }
```

You should be able to see that part where this transition is picked up, and the kLandAnim animation gets played.

```
private var m_climbLadder:Boolean;
```

This is set when the player is confirmed as climbing the ladder; in this event a special case control mechanism will take over from the normal gravity/friction behaviour which actually enables the climbing itself.

```
private var m_onLadderTop:Boolean;
```

This is set by the collision system when the player is confirmed as being on the top of a ladder - this has a dual purpose; it lets me have a special case so I can apply the same collision logic as I would do when standing on a jump-through platform and it also allows me to detect the down arrow in the player input control mechanism as an indication that the player wants to join the ladder at the top.

```
private var m_disableLadderTopCollision:Boolean;
```

This is set when joining the ladder at the top and allows the collision system to ignore the ladder top tile until the player is confirmed as having his centre point in the AABB of any ladder tile. This causes the player to fall briefly down until they start colliding with the middle of the ladder tile, where collision is then re-enabled.

Special ladder physics

I covered the regular collision response code in the last article (the part where friction is applied etc). When m_climbLadder is set, this code is ignored and some new code takes over.

The player has a special variable called m_velTarget which is 0,0 by default and only comes into use when we're confirmed as being on a ladder. This gets set to the maximum player speed in the X or Y axis inside the player input system when the player is moving left/right or up/down and is on a ladder.

The main body of the code tries to alter the player's actual velocity (which is Player.m_vel) towards this new target velocity, and to prevent the player from arriving at the target velocity too quickly there is a special Player.kLadderStayStrength constant. This prevents the ladder movement from contrasting with the player's movement in the rest of the game.

The full code looks like this:

```
package Code.Characters { ... public class Player extends Character { /// <summary> ///
Special case code for the player colliding with different tile types /// </summary> protected
override function InnerCollide(tileAabb:AABB, tileType:int, dt:Number, i:int, j:int ):void
{ ... else if ( Map.IsTileLadder( tileType ) ) { // // Are we colliding with the top of a
ladder? // var checkLadderMiddle:Boolean = false; if ( Map.IsTileLadderTop( tileType ) && !
m_disableLadderTopCollision ) { m_onLadderTop = Collide.AabbVsAabbTopPlane( this, tileAabb,
m_contact, i, j, m_map ); if ( m_onLadderTop ) { CollisionResponse( m_contact.m_normal,
m_contact.m_dist, dt ); } else { checkLadderMiddle = true; } } else { checkLadderMiddle =
true; } if (checkLadderMiddle) { // // check to see if we're colliding with the middle of a
ladder // if (Collide.PointInAabb(m_Pos, tileAabb)) { m_collideLadder = tileType;
m_disableLadderTopCollision = false; if ( m_climbLadder ) { // remove a portion of the total
velocity of the character var delta:Vector2 = m_velTarget.Sub( m_vel ); var len:Number =
delta.m_Len; var change:Vector2 = delta; if ( len>kLadderStayStrength ) { // limit the amount
we can remove velocity by change.MulScalarTo( kLadderStayStrength/len ); }
m_vel.AddTo( change ); } } } } ... } } }
```

The relevant part is at the bottom.

What's going on in the above code? Well, once we're inside the condition:

```
if ( m_climbLadder ) {
```

then we're actually doing the ladder physics. First of all we work out the difference between our current velocity and the desired velocity:

```
var delta:Vector2 = m_velTarget.Sub( m_vel );
```

Then, if the length of this vector is longer than our ladder stay strength amount (i.e. the movement is too much to do in one frame), we have to clamp the movement so its just as much as we're allowed to move by the constant kLadderStayStrength:

```
var change:Vector2 = delta; if ( len>kLadderStayStrength ) { // limit the amount we can
remove velocity by change.MulScalarTo( kLadderStayStrength/len ); }
```

Once this has been clamped we add this delta to our velocity vector and we're done:

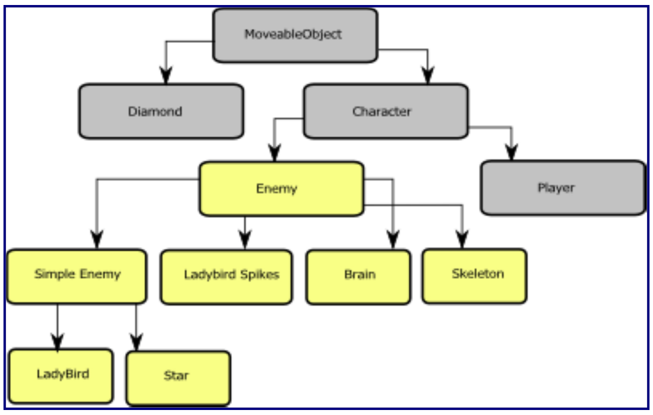
```
m_vel.AddTo( change );
```

Once the player stops colliding with a ladder tile, the normal collision resolution code takes over again and we're done with ladders completely!

The joy of AI

I really enjoyed working on the AI for this demo because it went so amazingly smoothly, nearly everything I wanted to try worked first time. It was refreshing after all the messing around with the ladder stuff; you learn to enjoy these small victories after you've been programming for a while because they're so rare 🙂

Lets have another look at the class hierarchy I showed in the last article:



AI Hierarchy

The complete AI hierarchy is shown as the yellow branch which inherits from Character.

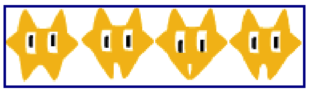
Enemy, the base class

Ok, so lets have a look at the base class which all enemies derive from:

```
package Code.Characters { import flash.display.*; import Code.Maths.*; import Code.*; import
Code.Geometry.*; import Code.Graphics.AnimationController; import Code.System.*; import
Code.Level.*; public class Enemy extends Character { // animation names private const
kHitAnim:String = "hitAnim"; /// <summary> /// Simple constructor /// </summary> public
function Enemy( ) { // aways go to the left by default; m_vel.m_x = -m_WalkSpeed;
m_animationController.PlayLooping( kWalkAnim );
m_animationController.GotoRandomFrameInCurrentAnim( ); } /// <summary> /// Simple update
function /// </summary> public override function Update( dt:Number ):void { if ( m_hurtTimer
> 0 ) { m_hurtTimer--; } super.Update( dt ); } /// <summary> /// Hurt this enemy, player was
at hurtPos /// </summary> public override function Hurt( hurtPos:Vector2 ):void { if
( m_hurtTimer==0 ) { m_hurtTimer = kEnemyHurtFrames; m_animationController.PlayOnce( kHitAnim
); m_animationController.SetEndOfAnimCallback( DeleteEnemy ); } } /// <summary> /// Animation
callback used to delete this enemy and spawn a pick-up /// </summary> private function
DeleteEnemy( animName:String ):void { // mark this enemy as dead, it will be deleted m_dead =
true; // spawn a pickup m_platformer.SpawnMo( DiamondPickupFla, m_pos.Clone( ), true ); } ///
<summary> /// What speed should this guy walk at? /// </summary> protected function get
m_WalkSpeed( ):Number { throw new NotImplementedException; } /// <summary> /// Does he kill
on touch? /// </summary> public function get m_KillsOnTouch( ):Boolean { throw new
NotImplementedException; } } }
```

It requires any concrete implementations to define a couple of attributes: m_WalkSpeed and m_KillsOnTouch. m_WalkSpeed is used in the constructor and as the name suggests is the speed at which the enemy walks. m_KillsOnTouch is used in the collision detection system and defines whether this particular enemy will kill the player on touch.

All enemies are required to provide a couple of animations with specific names which are referenced in this base class (or in Character from which this derives):



Star walk anim

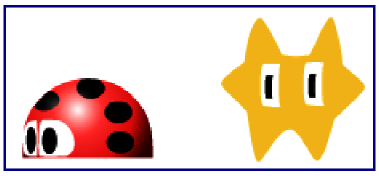
- 'walkAnim' - the walk animation this enemy has
- 'hitAnim' - the animation to play when this enemy is punched by the player

The individual enemy types may well provide additional animations, but the above is the bare minimum.

The other function worth mentioning is Hurt() which is actually defined in Character. This is overridden here with a custom implementation which counts down a few frames, then triggers the 'hitAnim' and attaches a callback which will happen when the animation is done playing; this marks the enemy for deletion and also spawns a pick-up where the enemy was at the time.

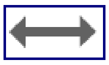
Simple enemy

This is another shared, non-concrete class which encompasses the behaviour of two of the enemy types.

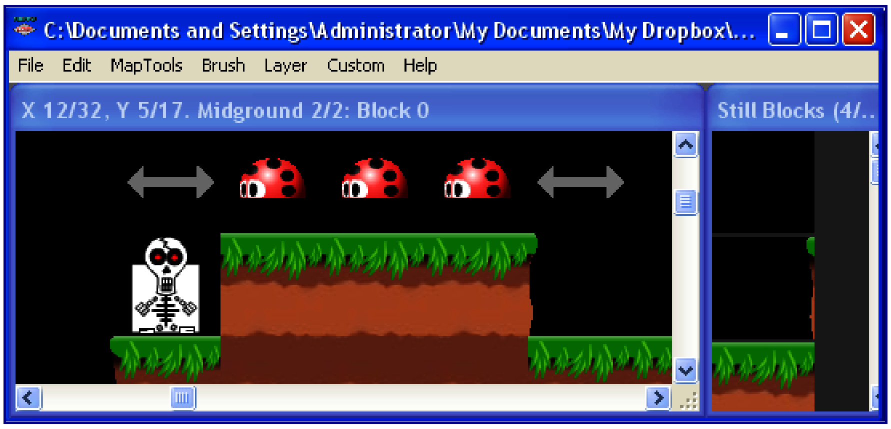

Simple enemies

```
package Code.Characters { import flash.display.*; import Code.Maths.*; import Code.*; import Code.Geometry.*; import Code.Graphics.AnimationController; import Code.System.*; import Code.Physics.*; public class SimpleEnemy extends Enemy { /// <summary> /// /// </summary> public function SimpleEnemy() { super(); } /// <summary> /// Simple update function which just checks for kReverseDirection markers /// </summary> public override function Update( dt:Number ):void { if ( m_hurtTimer == 0 ) { // form AABB for character var min:Vector2 = m_pos.Sub( m_halfExtents ); var max:Vector2 = m_pos.Add( m_halfExtents ); m_map.DoActionToTilesWithinAabb( min, max, InnerCollide, dt ); } super.Update( dt ); } /// <summary> /// Is the current tile a reverse direction maker? If so, change direction /// </summary> protected override function InnerCollide( tileAabb:AABB, tileType:int, dt:Number, i:int, j:int ):void { if ( tileType==eTileTypes.kReverseDirection && MoveableObject.HeadingTowards(tileAabb.m_Centre, this) ) { // toggle direction m_vel.m_x *= -1; this.scaleX *= -1; } } /// <summary> /// This enemy type is always updated, this is to stop them from bunching up /// due to the off-screen deactivation code /// </summary> public override function get m_ForceUpdate( ):Boolean { return true; } } }
```

Their basic behaviour is to head in a constant direction until they encounter a special marker type which indicates they should reverse direction. This marker type is always mapped in the middle collision layer in the map along with all other AI stuff.


Change direction marker

The marker itself gets a special tile (shown above) so the mapper (i.e me) can see where I've placed it in the map. At runtime, these tiles are turned invisible, or rather they don't actually have any visual data exported with them.


C:\Documents and Settings\Administrator\My Documents\My Dropbox\...
File Edit MapTools Brush Layer Custom Help
X 12/32, Y 5/17. Midground 2/2: Block 0
Still Blocks (4/..

AI markers as seen in Mappy

The above shows them in a level in Mappy.

The relevant code is for this behaviour is:

```
/// <summary> /// Simple update function which just checks for kReverseDirection markers /// </summary> public override function Update( dt:Number ):void { if ( m_hurtTimer == 0 ) { // form AABB for character var min:Vector2 = m_pos.Sub( m_halfExtents ); var max:Vector2 = m_pos.Add( m_halfExtents ); m_map.DoActionToTilesWithinAabb( min, max, InnerCollide, dt ); } super.Update( dt ); }
```

...which forms an AABB for the extents of the character and then calls out to the collision system to return all tiles which intersect with this AABB. The collision system then calls back to my callback function InnerCollide:

```
/// <summary> /// Is the current tile a reverse direction maker? If so, change direction /// </summary> protected override function InnerCollide( tileAabb:AABB, tileType:int, dt:Number, i:int, j:int ):void { if ( tileType==eTileTypes.kReverseDirection && MoveableObject.HeadingTowards(tileAabb.m_Centre, this) ) { // toggle direction m_vel.m_x *= -1; this.scaleX *= -1; } }
```

...this checks the tile type being collided with is of type eTileTypes.kReverseDirection, if so and if the enemy is heading towards this tile then we toggle the enemy's direction and also have him face the opposite way. That last condition (which checks the enemy's direction) is very important, otherwise it could be that the enemy is still going to be colliding with the same change direction marker next frame; this would lead to a very confused enemy who would toggle direction constantly.

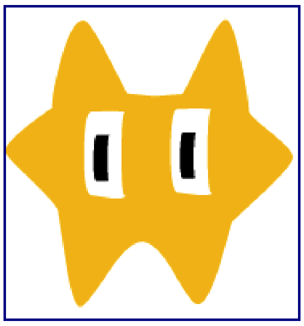
For reference, here is the utility function to work out if an object is heading towards a point (in the X axis):

```
/// <summary> /// Is the given candidate heading towards towardsPoint? /// </summary> static public function HeadingTowards( towardsPoint:Vector2, candidate:MoveableObject ):Boolean { var deltaX:Number = towardsPoint.m_x-candidate.m_Pos.m_x; var headingTowards:Boolean = deltaX*candidate.m_Vel.m_x>0; return headingTowards; }
```

I've found this useful in a couple of places.

Star

This is where we really start to feel the benefit of all this abstraction.



Star

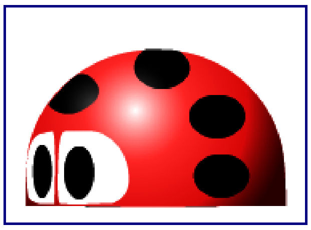
Here is the complete implementation of this enemy:

```
package Code.Characters { public class Star extends SimpleEnemy { private const
kWalkSpeed:Number = 40; private const kWalkAnimMultiplier:Number = 0.125; ///
<summary> /// /// </summary> public function Star() { super(); } /// <summary> /// Simple
accessor /// </summary> protected override function get m_WalkSpeed( ):Number { return
kWalkSpeed; } /// <summary> /// Simple accessor /// </summary> protected override function
get m_AnimSpeedMultiplier( ):Number { return kWalkAnimMultiplier; } /// <summary> /// Kills
on touch /// </summary> public override function get m_KillsOnTouch( ):Boolean { return true;
} } }
```

As you can see, he is nothing more than three simple accessors which don't even require an explanation! All the functionality is performed by the classes he derives from.

Ladybird

This one is a little more complex because he has a special behaviour where he pauses, plays an anim, emits another object, player another anim and then resumes his original motion.



Ladybird

Side note: I have no idea why we call this creature a 'ladybird' in the uk - logic dictates that 'ladybug' (the american name) is surely more fitting!

```
package Code.Characters { import Code.Maths.*; import Code.Physics.*; public class LadyBird
extends SimpleEnemy { // animation names private const kFireInAnim:String = "fireInAnim";
private const kFireOutAnim:String = "fireOutAnim"; private const kWalkSpeed:Number = 40;
private const kWalkAnimMultiplier:Number = 1; private const kFireSpikesDelay:int = 60;
private const kTriggerTimerRadius:Number = 200; private var m_fireSpikesCounter:int; private
var m_originalXVel:Number; /// <summary> /// /// </summary> public function LadyBird()
{ super(); m_fireSpikesCounter = kFireSpikesDelay; } /// <summary> /// /// </summary> public
override function Update( dt:Number ):void { super.Update( dt ); var withinRadius:Boolean =
m_platformer.m_Player.m_Pos.Sub( m_pos ).m_Len<kTriggerTimerRadius; var facingPlayer:Boolean
= MoveableObject.HeadingTowards( m_platformer.m_Player.m_Pos, this ); if (withinRadius &&
facingPlayer && m_hurtTimer==0 ) { // every so often, we spawn some spikes if
( m_fireSpikesCounter>0 ) { m_fireSpikesCounter--; } else if (m_animationController.m_Playing
== kWalkAnim) { // begin a series of callbacks which eventually plays the spawn anim
m_animationController.StopAtEnd( );
m_animationController.SetEndOfAnimCallback( WaitTillEndOfAnim ); } } } /// <summary> /// ///
</summary> private function WaitTillEndOfAnim( animName:String ):void
{ m_animationController.PlayOnce( kFireInAnim );
m_animationController.SetEndOfAnimCallback( SpawnSpikes ); // pause motion m_originalXVel =
m_vel.m_x; if ( m_hurtTimer==0 ) { m_vel.m_x = 0; } } /// <summary> /// /// </summary>
private function SpawnSpikes( animName:String ):void { var velX:Number =
-this.scaleX*kWalkSpeed*2; var spikes:Character =
Character(m_platformer.SpawnMo( LadySpikesFla, m_pos.Clone(), true, velX )); // put behind us
in display list order m_platformer.setChildIndex( spikes, m_platformer.getChildIndex( this )-
1 ); m_animationController.PlayOnce( kFireOutAnim );
m_animationController.SetEndOfAnimCallback( ResumeNormalAnim ); } /// <summary> /// ///
</summary> private function ResumeNormalAnim( animName:String ):void
{ m_animationController.PlayLooping( kWalkAnim ); // reset counter m_fireSpikesCounter =
kFireSpikesDelay; // resume motion if ( m_hurtTimer==0 ) { m_vel.m_x =
m_originalXVel; } } /// <summary> /// /// </summary> protected override function get
m_WalkSpeed( ):Number { return kWalkSpeed; } /// <summary> /// /// </summary> protected
override function get m_AnimSpeedMultiplier( ):Number { return kWalkAnimMultiplier; } ///
<summary> /// /// </summary> public override function get m_KillsOnTouch( ):Boolean { return
false; } } }
```

Lets have a look at the main update loop:

```
/// <summary> /// /// </summary> public override function Update( dt:Number ):void
{ super.Update( dt ); var withinRadius:Boolean =
m_platformer.m_Player.m_Pos.Sub( m_pos ).m_Len<kTriggerTimerRadius; var facingPlayer:Boolean
= MoveableObject.HeadingTowards( m_platformer.m_Player.m_Pos, this ); if (withinRadius &&
facingPlayer && m_hurtTimer==0 ) { // every so often, we spawn some spikes if
( m_fireSpikesCounter>0 ) { m_fireSpikesCounter--; } else if (m_animationController.m_Playing
== kWalkAnim) { // begin a series of callbacks which eventually plays the spawn anim
m_animationController.StopAtEnd( );
m_animationController.SetEndOfAnimCallback( WaitTillEndOfAnim ); } } }
```

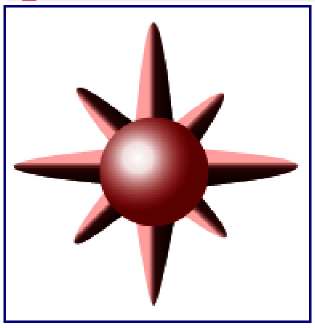
So, what's going in here? Firstly we do some simple maths to determine whether the player is within some radius of the enemy. We also work out if the enemy is facing the player (after all, no point trying to attack the player if he's behind you). If both of these conditions are true we decrement a counter, once this counter gets to 0, we trigger a sequence of animations via callbacks:

The first simply waits until the end of the current anim - this is to make sure the animation frames match up; since I designed the anims to follow on from each other.

```
/// <summary> /// /// </summary> private function WaitTillEndOfAnim( animName:String ):void {
m_animationController.PlayOnce( kFireInAnim );
m_animationController.SetEndOfAnimCallback( SpawnSpikes ); // pause motion m_originalXVel =
m_vel.m_x; if ( m_hurtTimer==0 ) { m_vel.m_x = 0; } }
```

When the above callback is called, we trigger the actual animation for spawning the spikes. This is done so the player knows to prepare themselves to get out of the way. I also pause the motion of the creature at this point, taking a note of what its original motion was so I can restore it afterwards.

```
/// <summary> /// /// </summary> private function SpawnSpikes( animName:String ):void { var
velX:Number = -this.scaleX*kWalkSpeed*2; var spikes:Character =
Character(m_platformer.SpawnMo( LadySpikesFla, m_pos.Clone(), true, velX )); // put behind us
in display list order m_platformer.setChildIndex( spikes, m_platformer.getChildIndex( this )-
1 ); m_animationController.PlayOnce( kFireOutAnim );
m_animationController.SetEndOfAnimCallback( ResumeNormalAnim ); }
```



Spawned spikes

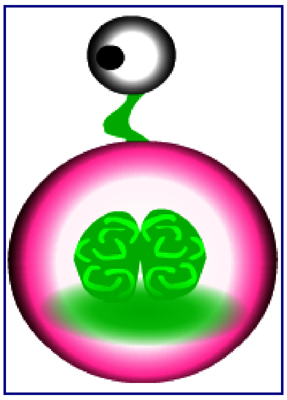
Once the spawn spikes animation has finished, I actually spawn a new object, which is another very simple class which obeys gravity and does full collision but also kills on touch. I make sure it heads in the same direction that the original enemy is facing, make sure it appears behind this enemy and lastly play the fired animation which sets up the synchronisation to match with the frame the normal animation ended on.

```
/// <summary> /// /// </summary> private function ResumeNormalAnim( animName:String ):void
{ m_animationController.PlayLooping( kWalkAnim ); // reset counter m_fireSpikesCounter =
kFireSpikesDelay; // resume motion if ( m_hurtTimer==0 ) { m_vel.m_x = m_originalXVel; } }
```

Once the previous animation has finished, the firing action is complete, so we resume playing the regular walk animation, reset the counter and restore the original motion of the enemy.

The brain

This enemy type has a pretty simple behaviour as well: it kills on touch and it will move for a while, then pause for a while and then carry on. When it starts to move it heads towards wherever the player was at the time.



The brain

```
package Code.Characters { import Code.Maths.*; public class Brain extends Enemy { private
const kWalkSpeed:Number = 80; private const kThinkSeconds:Number = 2; private const
kMoveSeconds:Number = 1; private var m_thinkTimer:Number; private var m_moveTimer:Number;
private var m_think:Boolean; public function Brain( ) { m_thinkTimer = Scalar.RandBetween( 0,
kThinkSeconds ); m_think = true; } /// >summary< /// Pause, then target the player and move
for a bit, repeat /// >/summary< public override function Update( dt:Number ):void
{ super.Update( dt ); if (!IsHurt()) { if ( m_think ) { // have we gone past our think timer
limit? if ( m_thinkTimer>0 ) { // toggle modes m_think = false; // work out a new target
velocity m_vel =
m_platformer.m_Player.m_Pos.Sub( m_pos ).UnitTo( ).MulScalarTo( kWalkSpeed ); // reset this
timer m_moveTimer = kMoveSeconds; } m_thinkTimer -= dt; } else { // have we gone past our
move timer limit? if ( m_moveTimer>0 ) { // toggle modes m_think = true; // stop moving
m_vel.Clear( ); // reset this timer m_thinkTimer = kThinkSeconds; } m_moveTimer -=
dt; } } } /// >summary< /// Simple accessor /// >/summary< protected override function get
m_WalkSpeed( ):Number { return kWalkSpeed; } /// >summary< /// Simple accessor /// >/summary<
public override function get m_KillsOnTouch( ):Boolean { return true; } } }
```

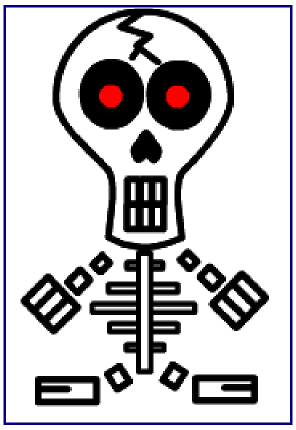
All the magic is within the Update function. As you can see there are a couple of simple timers which tick down depending on what mode the enemy is currently in (either m_think==true or not). When the think timer becomes less than 0, the enemy toggles modes and picks a direction to move.

```
// work out a new target velocity m_vel =
m_platformer.m_Player.m_Pos.Sub( m_pos ).UnitTo( ).MulScalarTo( kWalkSpeed );
```

This new direction takes the form of a velocity. This velocity is based on the vector from the enemy to the player, which is then made unit length and scaled by the kWalkSpeed of the enemy. This will cause the enemy to move in the direction of the player at a constant speed. Once he gets there he kills on touch.

The skeleton

This is the final AI character and has the most complex behaviour of them all.

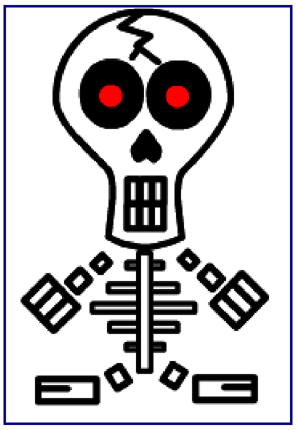


Skeleton

Its all driven via the animations though, which makes it rather flexible.

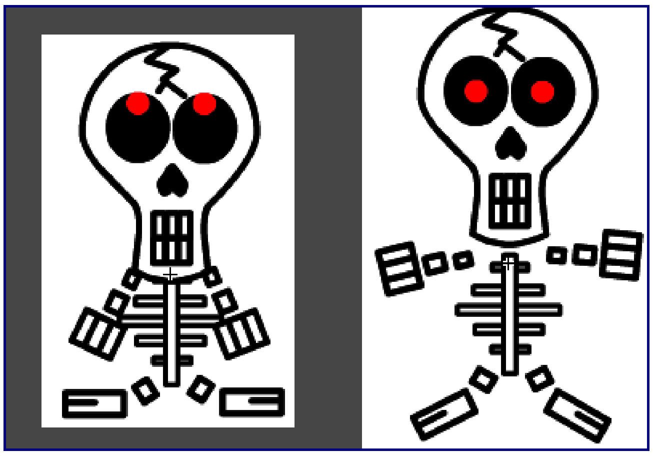
```
package Code.Characters { import Code.Maths.Vector2; public class Skeleton extends Enemy { //
animation names private const kJumpAnim:String = "jumpAnim"; private const kDeadlyAnim:String
= "deadlyAnim"; private const kJumpStrength:Number = 800; private var m_jumping:Boolean;
public function Skeleton( ) { super( ); ResetJumpSequence(null); } /// >summary< /// Stand
still, play walk anim /// >/summary< private function
ResetJumpSequence( animName:String ):void { // stand still m_vel.m_x = 0; m_jumping = false;
m_animationController.PlayOnce( kWalkAnim );
m_animationController.SetEndOfAnimCallback( PlayJumpAnim ); } /// >summary< /// Play the jump
anim /// >/summary< private function PlayJumpAnim( animName:String ):void
{ m_animationController.PlayOnce( kJumpAnim );
m_animationController.SetEndOfAnimCallback( JumpTowardPlayer ); } /// >summary< /// Actually jump towards the player /// >/summary< private function JumpTowardPlayer( animName:String ):void { if ( !IsHurt( ) ) { // work out a new target velocity var targetPos:Vector2 = m_platformer.m_Player.m_Pos.Add( Vector2.RandomRadius( 100 ) ); var playerUnitDirection:Vector2 = targetPos.SubFrom( m_pos ).UnitTo( ); // must jump upwards! playerUnitDirection.m_y = -2; // re-normalise playerUnitDirection.UnitTo( ); // jump towards player m_vel.AddTo( playerUnitDirection.MulScalarTo( kJumpStrength ) ); m_jumping = true; m_animationController.SetEndOfAnimCallback( null ); } } /// >summary< /// Check for landing, play the deadly animation when landed /// >/summary< public override function Update( dt:Number ):void { super.Update( dt ); if ( m_jumping && !IsHurt() ) { if ( m_OnGround&&!m_OnGroundLast ) { // landed! m_animationController.PlayOnce( kDeadlyAnim ); m_animationController.SetEndOfAnimCallback( ResetJumpSequence ); } } } /// >summary< /// Apply collision detection only when not hurt /// >/summary< public override function get m_HasWorldCollision( ):Boolean { return !IsHurt(); } /// >summary< /// Apply gravity only when not hurt /// >/summary< protected override function get m_ApplyGravity( ):Boolean { return !IsHurt(); } /// >summary< /// Apply friction only when not hurt /// >/summary< protected override function get m_ApplyFriction( ):Boolean { return !IsHurt(); } /// >summary< /// Doesn't walk /// >/summary< protected override function get m_WalkSpeed( ):Number { return 0; } /// >summary< /// Force update when jumping so he's not left in mid air when going off screen /// >/summary< public override function get m_ForceUpdate( ):Boolean { return super.m_ForceUpdate || m_jumping; } /// >summary< /// Only kills on touch when the deadly anim is playing /// >/summary< public override function get m_KillsOnTouch( ):Boolean { return m_animationController.m_Playing==kDeadlyAnim; } } }
```

Like in the case of the LadyBird there are a number of animations which all lead on from each other when played in sequence:



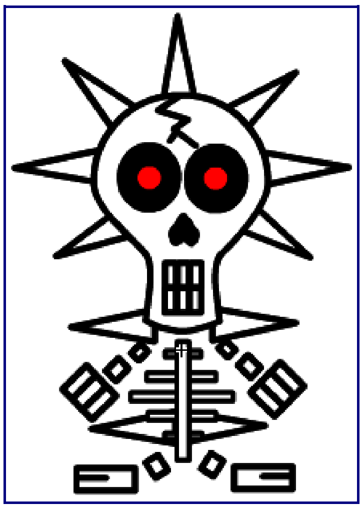
Walk anim

(His eyes just move left to right)



Prepare and then jump anim

(Prepares to and then finally jumps - to give the player fair warning)



Lands anim

(Lands and deadly spikes protrude - he will kill on touch at this point)

The only part of the code which needs a special mention is where the skeleton is choosing which direction to jump in:

```
/// >summary< /// Actually jump towards the player /// >/summary< private function
JumpTowardPlayer( animName:String ):void { if ( !IsHurt( ) ) { // work out a new target
velocity var targetPos:Vector2 = m_platformer.m_Player.m_Pos.Add( Vector2.RandomRadius( 100 )
); var playerUnitDirection:Vector2 = targetPos.SubFrom( m_pos ).UnitTo( ); // must jump
upwards! playerUnitDirection.m_y = -2; // re-normalise playerUnitDirection.UnitTo( ); // jump
towards player m_vel.AddTo( playerUnitDirection.MulScalarTo( kJumpStrength ) ); m_jumping =
true; m_animationController.SetEndOfAnimCallback( null ); } }
```

The code picks a random position around the player:

```
// work out a new target velocity var targetPos:Vector2 =
m_platformer.m_Player.m_Pos.Add( Vector2.RandomRadius( 100 ) );
```

This was done to stop multiple skeletons from landing in the exact same spot too much, which looked unnatural. Then, this target position is turned into a unit length direction vector, which points from the skeleton towards the player:

var playerUnitDirection:Vector2 = targetPos.SubFrom(m_pos).UnitTo( );

But of course when jumping, one must always jump upwards, so I simply set the y axis of the vector to be -2. This then breaks the unit length constraint, so I have to renormalise again:

```
// must jump upwards! playerUnitDirection.m_y = -2; // re-normalise
playerUnitDirection.UnitTo( );
```

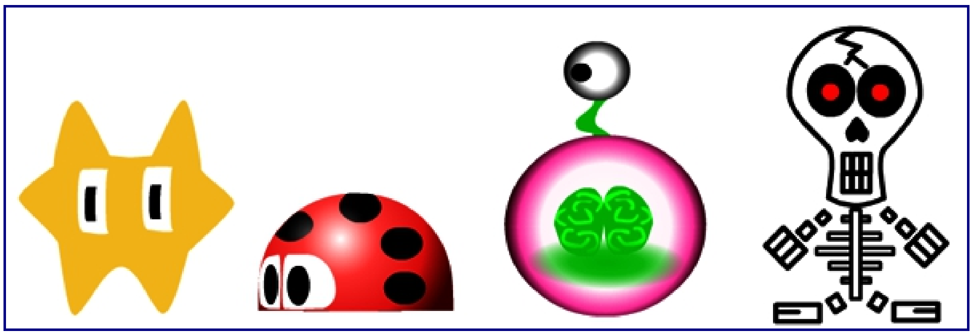
The reason it was -2 and not -1 or some other number is that this forces the final vector to be mostly pointing upwards with just a little side to side motion. If I'd chosen -1, it would have been a more even spread.

You can use normalisation like this to create different spread patterns, whereby a random unit vector is added to a known fixed direction vector, then renormalised. I've used this technique in the past to create an emission spread for a particle system which needed to emit particles in a cone formation; it was random point in a unit sphere plus some predefined forward direction (scaled as appropriate) and then renormalised.

Then I actually perform the jump by setting the skeleton's velocity and clear the animation callback to reset the sequence:

```
// jump towards player m_vel.AddTo( playerUnitDirection.MulScalarTo( kJumpStrength ) );
m_jumping = true; m_animationController.SetEndOfAnimCallback( null );
```

And that's it!



The motley crew

That concludes my series on how to make a 2d platformer, I hope you've enjoyed reading it!

As ever, if you want, you can buy the source-code for the entire game (or even try a version **for free**), including all the assets and levels you see above. It will require Adobe Flash CS4+, the Adobe Flex Compiler 4.0+ and either Amethyst, or Flash Develop to get it to build. And you'll want Mappy or some alternative in order to create your own levels!

Following on from feedback from the Angry Birds article, I've included a Flash Develop project as well as an Amethyst project inside the .zip file, to help you get started more quickly, no matter which development environment you have.

You are free to use it for whatever purposes you see fit, even for commercial games or in multiple projects, the only thing I ask is that you don't spread/redistribute the source-code around. Please note that you will need some programming and design skills in order to make the best use of this!

Go to the source-code option page to choose the version you'd like - from **completely free** to the full version!

Subscribers can access the source here

Until next time, Have fun!

Cheers, Paul.

How to make a Platform Game - source-code options

Following some feedback on reddit regarding the price of the source code, I've decided to offer people who'd like to get access to the source code accompanying this series of articles a few choices.

Totally free version

Click to give focus, then cursor keys control the green player character.

This versions contains the bare essentials of the game, one player character, one map, one tile type, no level progression, no enemies, no fancy graphics, or dependence on the Flash IDE. Basically, everything you see in the demo above, including the nice robust camera system.

This version is recommended for anyone who wants to get a pretty decent code framework in which to build their vision of the game or for anyone who is interested in sampling the quality of my code before making a purchase of the other, more advanced versions.

Download it here for free

Update: this version is now available on IOS, courtesy of Gabriel Sanchez who wrote an article about the port. You can read it (and download his version) here: http://gabrieldsblog.wordpress.com/2013/12/29/2d-platform-game-ios-port-part-1/

USD 9.99 version

This version contains all the assets of the full game, but only one enemy type, no level progression, no parallax layers and no ladders.

I'd recommend this version for those who are interested primarily in having a very solid framework with a good set of features, have a strong desire to put their own mark on the game and a willingness to experiment and learn.

USD 9.99

USD 19.99 version - SALE: reduced from $49.99!

This is the complete version, with everything included.

I'd recommend this version for anyone who wants a nearly complete game they can make their own. Just by re-doing the gfx, and adding more levels you will have a completely unique and fun game.

USD 19.99

USD 49.99 version

This is the complete version, with graphics redone by Khan Studios.

I'd recommend this version for anyone who wants a neat tile-set and animated character set to go with the project, and to use in whatever else they see fit.

USD 49.99